Compiling Abstract Specifications into Concrete Systems – Bringing Order to the Cloud

Ian Unruh
Kansas State University
iunruh@ksu.edu

Alexandru G. Bardas
Kansas State University
bardasag@ksu.edu

Rui Zhuang
Kansas State University
zrui@ksu.edu

Xinming Ou
Kansas State University
xou@ksu.edu

Scott A. DeLoach
Kansas State University
sdeloach@ksu.edu

Abstract

Currently, there are no suitable abstractions that allow cloud users to define the structure and dependency of the services in a cloud-based IT system. As a result, cloud users either have to manage the low-level details of the cloud services directly, such as in IaaS, or resort to SaaS or PaaS where much or part of the services are managed by the cloud provider but are less flexible to customize to better suit user needs. We propose a high-level abstraction called the *requirement model* for defining cloud-based IT systems. It captures the important aspects of a system's structure, such as service dependencies, without considering the low-level details such as operating systems or application configurations. The requirement model separates the cloud customer's concern of *what* the system does, from the system engineer's concern of *how* to implement it. We further develop a "compilation" process that automatically translates a requirement model into a concrete system based on pre-defined and reusable knowledge units. This higher-level specification and the associated compilation process allows repeatable deployment of cloud-based IT systems, supports more reliable system management, and enables implementing the same requirement in multiple ways. We demonstrate the practicality of this approach in the *ANCOR* (Automated eNterprise network COmpileR) framework, which takes a requirement model and generates an IT system based on that specification. Our current implementation targets OpenStack and uses Puppet to configure the cloud instances, although it could work with other cloud platforms and configuration management solutions as well.

1 Introduction

Cloud computing is revolutionizing industry, reshaping the way IT systems are designed, deployed and utilized. However, with every revolution comes problems. Already, companies that have moved resources into the cloud are using terms like "virtual sprawl" to describe the mess they have created [33]. Cloud services are currently offered in several models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). While these options allow customers to decide *how much* management they want to perform for their cloud-based systems, they do not provide good *abstractions* for effectively managing them and addressing diverse user needs.

IaaS (*e.g.*, Amazon Web Services and OpenStack) allows cloud users to access the raw resources (compute, storage, bandwidth, *etc.*); however, it forces users to manage the software stack in their cloud instances at a low level. While this approach gives users tremendous flexibility, it also allows the users to create badly configured or misconfigured systems, raising significant concerns (especially related to security) [11, 13]. Moreover, offering automatic scalability and failover is challenging for cloud providers because the replication and state management procedures are application-dependent [9]. On the other hand, SaaS (also known as "on-demand software") provides pre-configured applications to cloud users (*e.g.,* SalesForce and Google Apps). Users typically choose from a set of predefined templates, which makes it difficult to adequately address the range of user needs. PaaS (*e.g.,* Google App Engine, Heroku, and Microsoft Azure) is somewhere in the middle, offering computing platforms with various pre-installed operating systems and software applications and allowing users to deploy their own software as well. While this is a trade-off between IaaS and SaaS, it also inherits the limitations of both to various degrees.

We observe that the limitations of the existing cloud service models are due to the lack of a *higher-level abstraction* for systems in a cloud. The abstraction should have the following properties.

1. The abstraction must capture *what* a cloud user needs instead of low-level details on *how* to implement those needs. A major economic motivation for

using cloud infrastructures is to outsource IT management to a more specialized workforce (called *system engineers* thereafter). Therefore, it is likely that cloud users would prefer to focus on communicating their needs in an abstract way as opposed to low-level configuration details.

2. There must be a process to automatically compile the abstraction into a valid concrete system. Such compilation should use well-defined knowledge units built by the system engineers and be capable of compiling the same abstract specification to different types of concrete systems based on low-level implementation choices.

We believe such an abstraction will benefit all the three cloud service models. For IaaS, the abstract specification will act as a common language between cloud users and system engineers to define the system the user wants, while the compilation process becomes a tool that enables system engineers to be more efficient in their jobs. Re-using the compilation knowledge units will also spread the labor costs of creating those units across a large number of customers. In the SaaS model the system engineers will belong to the cloud provider so the abstract specification and the compilation process will help them provide better service at a lower cost. In the PaaS model we foresee cloud vendors providing an initial set of abstract specifications that users and system engineers can extend to meet their needs and then compile into concrete systems.

In our review of the literature and commercial/open-source products, we failed to find any existing systems that can achieve the above vision. While some companies provide solutions that address cloud management challenges, these solutions (*e.g.,* AWS OpsWorks) tend to focus on automation as opposed to abstraction. Current solutions include scripts that automatically create virtual machines, install software applications, and manage the machine/software lifecycle, while others are able to dynamically scale the computing capacity [32, 33, 34]. Unfortunately, none of these solutions provide a way to explicitly document the dependencies between the deployed applications. Instead, dependencies are *inferred* using solution-specific methods (*e.g.,* Chef [30] and Puppet [26]) for provider-specific platforms. Therefore, the lack of a consistent model for configuring the cloud creates a number of problems: network deployments and changes cannot be automatically validated, automated solutions are error-prone, incremental changes to networks are challenging (if not impossible) to automate, and network configuration definitions are unique to specific cloud providers and are not easily ported to other providers.

To achieve our vision, we propose a model that separates user requirements from the actual implementation by creating a thin layer of abstraction between the two. In other words, we developed an abstract, domain-specific language that enables cloud customers to specify their requirements abstractly without worrying about the underlying implementation. Then, this abstract specification can be automatically compiled into a concrete cloud-based system that meets the user requirements. Our abstract language explicitly captures dependencies among components of the system, improving maintainability. It also allows the system (or parts of it) to be recompiled based on different implementation choices, creating more flexibility. Furthermore, having an abstract, platform-independent system specification makes it easier to port a system between different cloud providers' infrastructures. The approach also enables us to address automatic scalability and failover even though these processes are highly application-dependent.

To demonstrate the efficacy of our approach, we implemented and evaluated a fully-functional prototype of our system, called *ANCOR* (Automated eNterprise network COmpileR). The current implementation of ANCOR is based on OpenStack [42]; however, the framework can also be targeted to other cloud platforms such as Amazon Web Services.

The rest of the paper is organized as follows: Section 2 explains some enabling technologies that we use in this work. Section 3 presents an overview of the framework along with the proposed abstract specification and the compilation workflow. In Section 4 we describe the implementation of the ANCOR framework and present a working real-world scenario. The future work is presented in Section 5, followed by our conclusions.

2 Enabling Technologies

Several new technologies have facilitated the development of our prototype. In particular, there has been several advancements in the configuration management technologies (CMT) that provide re-usable plugins to help streamline the configuration management process. This is especially beneficial to our work, since those plugins are the perfect building blocks for our compilation process. To help the reader better understand our approach, this section presents a basic background on the state-of-the-art CMTs.

Two widely used configuration management solutions are Chef [30] and Puppet [26]. We use Puppet in our work but similar concepts are also applicable to Chef. Puppet works by installing an agent on the host to be managed, which communicates with a controller (called the master) to receive configuration directives. The directives are written in a declarative language called Puppet *manifests*, which define the *desired configuration*

```
<% master = import(:database_master, :querying) %>
<% slaves = import_all(:database_slave, :querying) %>
<% redis = import(:kvstore, :redis) %>
---
classes:
  ancor::webapp:
    http_port: <%= resource(:http).port %>
    db_master_host: <%= master.ip_address %>
    db_master_port: <%= master.port %>
    db_slaves:
    <% slaves.each do |slave| %>
      - name: <%= slave.name %>
        host: <%= slave.ip_address %>
        port: <%= slave.port %>
    <% end %>
    redis_host: <%= redis.ip_address %>
    redis_port: <%= redis.port %>
    redis_poolsize: 12
    db_name: ancor_rails
    db_user: ancor_rails
    db_password: &5ab_3df90H
```

Figure 1: Fragment for a Web Application

state of the host (*e.g.*, installed packages, configuration files, running services, *etc.*). If the host's current state is different than the manifest received by the Puppet agent, the agent will issue appropriate commands to bring the system into the specified state.

In Puppet, manifests can be reused by extracting the directives and placing them in *classes*. Puppet classes accept parameters that are used to separate the configuration data (*e.g.*, IP addresses, port numbers, version numbers, *etc.*) from the configuration logic. Classes can be packaged together in a Puppet module for reuse. Typically, classes are bound to nodes in a master manifest known as the *site manifest*. Puppet also can be configured to use an external program to provide the classes that should be assigned to a node. This external program is known as an *External Node Classifier* (ENC). The format expected by the Puppet master from the ENC is YAML [17].

When the ENC format is combined with *Embedded Ruby* (ERB), it opens up the possibility for injecting values that are used as parameters for Puppet classes. This can be done each time a node requests its manifest from the Puppet master. We refer to the combination of the ENC output format and ERB as *fragments*. Figure 1 is an example fragment used to assign the Puppet class `ancor::webapp` to a node. It requires several parameters (`http_port`, `db_master_host`, etc.) that are calculated at run time by the ENC. This allows us to calculate those parameters based on an up-to-date model of the system, as opposed to hardcoding them into the manifests. We use this technology in the compilation process described later.

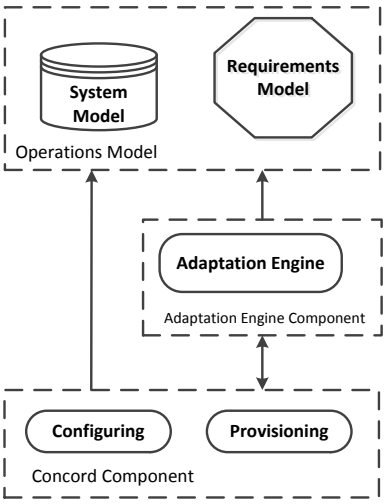

Figure 2: High-level View of the ANCOR Framework

3 The ANCOR Framework

Our requirements specification approach and the implemented framework (ANCOR) offer system engineers the same flexibility as in a typical IaaS model. This means that engineers can keep their workflow using their preferred configuration management tools (*e.g.,* Puppet and Chef) and orchestration tools (*e.g.,* MCollective). They have the option to do everything in their preferred ways up to the point where they connect the components (services) together. For example, system engineers have the option of using predefined configuration modules and leveraging the contributions from the CMT community. Or they can write their own manifests or class definitions to customize the system in their own ways. ANCOR can leverage all of these and does not force the system engineers to use particular low-level tools or languages; rather it provides the ability to manage the system based on a high-level abstraction, which currently does not exist in the available technologies.

There are three major components in the ANCOR framework (Figure 2): the operations model, the adaptation engine, and the Concord component. The direction of arrows in the diagram means referencing/updating. The operations model has two parts: the requirement model which captures user's requirements in an abstract way, and the system model which correponds to the concrete system in the cloud and is populated by the Concord component. The adaptation engine performs the compilation by referencing the requirement model, making implementation decisions necessary to satisfy the requirements, and instructing the Concord component to carry out the provisioning and configuration of the instances (virtual machines in the cloud). It can also instruct the Concord component for performing configuration changes when requested by users, while ensur-

ing the concrete system always satisfy the requirement model at all times. The Concord component has two subcomponents: provisioning and configuring. They are responsible for interacting with the configuration manager tool(s), orchestration tool(s), and the cloud-provider API.

The ANCOR framework manages the lifecycle of each instance in relation to dependent instances or clusters of instances. This management involves creating and deleting instances, adding and removing instances to and from clusters, and keeping dependent instances/clusters aware of the updates performed. The ANCOR framework simplifies network management as system dependencies are formalized and automatically enforced. Moreover, network resilience for traditional failures can also be addressed. We next explain the details of the requirement model and the workflow of compiling the model into a concrete system, as well as how to utilize the model to better manage the concrete system.

3.1 Requirement Model

A key idea behind our approach is to abstract the definition and structure of IT services into a model that can be used to generate and manage the concrete systems. We call this abstraction the *requirement model*. We also maintain the details of the concrete system in the *system model* for book-keeping purposes. The combined requirement model and system model is called the *operations model*. When ANCOR compiles a requirement model into a concrete system in the cloud, the system model is populated with the details of the cloud instances and their correspondence to the high-level model. When the system changes, the system model is always updated to ensure a consistent and accurate view of the system at all times. The detailed entity-relation diagram of the two models can be found in Figure 8 in the Appendix.

3.1.1 The ARML language

We specify the requirement model in a domain-specific language called the ANCOR Requirement Modeling Language (ARML). ARML's concrete syntax is based on YAML, which is a language that allows one to specify arbitrary key-value pairs. We show the *abstract syntax* of ARML in Figure 3. We use the notion $A \Rightarrow B$ to represent the key-value pairs specified in YAML. Upper-case identifiers represent non-terminal symbols, lower-case identifiers represent terminal symbols, and ARML keywords are distinguished by bold font.

A requirement model is composed of the specifications of *goals* and *roles*. A *goal* is a high-level business goal (*e.g.,* blog website, eCommerce website, *etc.*) and its purpose is to organize the IT resources around business objectives. A *role* defines a *logical unit of configuration*. A role may depend on/be depended on by other

ReqModel ::= **goals** ⇒ *GoalSpec*$^+$
roles ⇒ *RoleSpec*$^+$

GoalSpec ::= goalID ⇒
[**name** ⇒ string]
roles ⇒ roleID$^+$

RoleSpec ::= roleID ⇒
[**name** ⇒ string]
[**min** ⇒ integer]
[**exports** ⇒ *ResourceSpec*$^+$]
[**imports** ⇒ *ImportSpec*$^+$]
implementations ⇒ *ImplementationSpec*$^+$

ResourceSpec ::= resourceID ⇒
(**type** ⇒ resourceTypeID**,** *ResourceAttr****)**

ResourceAttr ::= attributeID ⇒ value

ImportSpec ::= roleID ⇒ resourceID$^+$

ImplementationSpec ::= implementationID ⇒ value

goalID, roleID, resourceID, attributeID, resourceTypeID, strategyID, implementationID, clusterID, tag are symbols. *integer* and *string* are defined in the usual way.

Figure 3: ARML Grammar

roles. Examples include a database role, a web application role, a message broker role, and so on. One can think of a role as representing a group of similarly configured instances that perform the same tasks. For example, if a company uses 100 instances to stand up its web front, the 100 instances are normally configured identically (except for network addresses) and a load balancer dispatches an incoming web request to one of the instances in the group. In our model we use a single role to represent all the instances that achieve the same functionality in the system. That is, we will have a single "web application" role for the 100 web server instances, and a "load balancer" role for the load balancer instance. The role-to-instance mapping is maintained in the "role-assignment" relation in the system model.

As an example, following is a goal specification for an eCommerce system written in ARML. The format of the specification is in YAML where each key-value pair $A \Rightarrow B$ is written as "$A : B$." The goal `ecommerce` has six roles supporting its mission. The role `weblb` is a load balancer for `webapp`, the web application role. There is a role for the database master (`database_master`) and the slave (`database_slave`), and so on.

```
goals:
  ecommerce:
    name: eCommerce frontend
    roles:
      - weblb
      - webapp
      - database_master
      - database_slave
```

```
    - workerapp
    - kvstore
```

Below is an example specification of the `weblb` and `webapp` role. A role uses a *resource* (a channel) to interact with other roles. A resource is an interface *exported* (provided) by a role and possibly *imported* (consumed) by other roles. Resources could include a single network port, a range of ports, or a Unix socket. For instance, the `webapp` role exports an `http` resource, which is a TCP port (*e.g.,* 80). The `weblb` role imports the `http` resource from the `webapp` role. A role is a "black box" to other roles, and only the exported resources are visible interfaces. Using these interfaces the requirement model captures the dependencies between the roles.

```
roles:
  weblb:
    name: Web application load balancer
    min: 2
    exports:
      http:  { type: single_port, protocol: tcp }
    imports:
      webapp: http

  webapp:
    name: Web application
    min: 3
    exports:
      http:  { type: single_port, protocol: tcp }
    imports:
      database_master: querying
      database_slave: querying
      kvstore: redis
```

The `webapp` role also imports three resources from various other roles: `querying` from `database_master`, `querying` from `database_slave`, and `redis` from `kvstore`. This means the `webapp` role depends upon three other roles: `database_master`, `database_slave`, and `kvstore`. The `min` field indicates the minimum number of instances that need to be deployed to play the role.

The requirement model also addresses instance clustering. A *cluster* is a collection of roles that work together to provide high-availability and/or load-balancing. For example, a company website with large amount of traffic normally will deploy a number of similarly configured web application instances to serve the web requests, and a load balancer dispatches a request to one of the target instances in the cluster. This cluster involves two roles: the load balancer and the web application server. The usual export/import fields in the role specifications will capture the dependency between the load balancer and the web application. Currently our model does not explicitly specify the fact that the two roles work together to form a web cluster. The dependency information captured in the export/import relationship is sufficient to support calculating configuration changes when, for example, a new web application instance will be added to the cluster and the load balancer needs to know about this to leverage the new resource. So far we have not seen real-world clustering strategies that require specifically modeling the cluster structure, beyond the dependency relationship between the roles that form the cluster. If such clustering strategies do turn out to exist, we will extend our requirement model to support this.

3.1.2 Role Implementation

Role names are application-specific and are chosen by the user or system engineers to convey the intution of what each role does in the system; there is no pre-defined role names in ARML (the only reserved names are those keywords shown in bold font in Figure 3). However, to automatically compile the requirement model into concrete systems, system engineers must provide the *semantics* for each role. This semantics is specified in the *implementation* field of the role specification, which defines how each instance must be configured to play that specific role. The Concord component then uses this implmentation information to properly configure and deploy the concrete instances. The value of the implementation field is thus dependent on the configuration management technology (CMT) being used to configure and deploy the system. This process is similar to traditional code compilation process for high-level programming languages, where abstact code constructs are compiled down to machine code. The compiler must contain the semantics of each code construct in terms of machine instructions for a specific architecture. As we draw this analogy between our ANCOR compiler and a programming language compiler, the natural question is "what is the architecture-equivalent of a cloud-based system?" In other words, is there a well-defined interface between the requirement model and the "cloud runtime" — the cloud platform and the CMT being used?

It turns out that a well-defined interface between the requirement model and the "cloud runtime" is well within reach if we leverage existing CMT technologies. As explained in Section 2, there has been a general movement in CMT towards encapsulating commonly-used configuration directives into reusable, parameterized modules. We use the term "fragment" to describe the code snippet that invokes a module with parameters derived from our high-level requirement model. These fragments then become the values of a role's "implementations" field. In addition, since there may be multiple ways to achieve a role's functionality, a role may have more than one implementation. In this case, it is up to the compiler to pick the appropriate role implementation for a specific concrete system as there will likely be compat-

```
<% backends = import_all(:webapp, :http) %>
---
classes:
  ancor::weblb:
    http_port: <%= resource(:http).port %>
    backends:
    <% backends.each do |be| %>
      - name: <%= be.name %>
        host: <%= be.ip_address %>
        port: <%= be.port %>
    <% end %>
```

Figure 4: Fragment for a Web Load Balancer

ibility constraints between the different implementations of the various system roles. (In our current prototype, we simply select the first role implementation specified in the "implementations" field and leave the constraint specification and implementation selection problems as future work.)

Figure 4 shows the implementation fragment for the `weblb` (or web load balancer) role. There are two parts in each fragment. The code before "`---`" extracts relevant values from the operations model such as information about the concrete instances assigned to specific roles. In this example, information about the set of backend web application instances is extracted and stored in the variable `backends`. The statement `import_all(:webapp, :http)` uses the operations model to find all instances that are assigned to play the role `webapp`, from which the load balancer imports the `http` resource. This information is then used to invoke a pre-defined Puppet class `ancor::weblb`, which takes two parameters: the HTTP port number and a list of backend web application instances along with their IP addresses and ports. The former can be found in the operations model as the concrete port number of the `http` resource exported by the load balancer (`resource(:http).port`), while the latter can be easily obtained from the `backends` variable (see Figure 1 for the implementation for the `webapp` role).

In summary, a fragment specifies *a concrete way to implement the intended functionality embodied by a role* by describing the invocation of pre-defined configuration modules with concrete parameters computed from the operations model. The use of a high-level requirement model that explicitly captures the dependencies among the various roles is crucial to automating this process. These fragments are not only useful when generating the system, but also for modifying as the system changes over time. For example, if a new instance is deployed to play the `webapp` role, the dependency structure in the operations model allows ANCOR to automatically discover all the other roles that may be impacted (those that depend on the `webapp` role) and "re-execute" their role fragments to ensure the configuration of each instance is consistent with the the operations model. This is the reason why we do not need to explicitly handle cluster management as it is handled like all other dependencies in the system.

We should also emphasize that while our current prototype uses Puppet, ANCOR can work with many mature CMT solution such as Chef, SaltStack [46], or CFEngine [40]. There are two important properties of all these mature CMT solutions: 1) the directives an agent receives dictates a *desired state* as opposed to commands for state changes, which allows configuration changes to handled in the same way as the initial configuration; 2) the solution provides a way for users to write pre-defined configuration modules (*e.g.*, Puppet classes) that can be reused in different systems. Such modules become the building blocks, or the "instruction set", into which ANCOR compiles the abstract requirement model.

3.2 The ANCOR Workflow

There are four main phases involved in creating and managing cloud-based systems using ANCOR.

1. Requirements model specification
2. Compilation choice specification
3. Compilation/Deployment
4. Maintenance

The first two phases result in the creation of the requirement model, and the next two phases perform the actual deployment and maintenance of the system in the cloud.

3.2.1 Requirement Model Specification

In this phase, the user and system engineers work together to define the goals of the system, which may require significant input from various stakeholders. Next, they determine the roles required to achieve each goal and the dependencies among the roles. This task could be handled by the system engineers alone or in consultation with the user. The high-level requirement language ARML provides an abstract, common language for this communication.

3.2.2 Compilation Choices Specification

In this phase, system engineers define role semantics using pre-defined CMT modules. In our current prototype this is accomplished by defining the fragments that invoke Puppet classes as described in Section 3.1.2. If no appropriate CMT modules exist, system engineers must define new fragments and place them into the repository for future use. In general, system engineers could specify multiple implementation choices for each role

to provide the ANCOR compiler flexibility in choosing the appropriate implmentation at runtime. One of the options available to system engineers is specifciation of the desired operating system for each instances. Here again, different operating systems may be used for each implementation of a role. With a wide variety of choices available to systems engineers, a constraint language is needed to specify the compatibility among the various implementation choices; we leave this for future research.

3.2.3 Compliation/Deployment

Once the requirement model has been defined, the framework can automatically compile the requirements into a working system on the cloud provider's infrastructure. This process has seven key steps:

1. The framework signals the adaptation engine to deploy a specific requirement model.
2. The adaptation engine makes several implementation decisions including the number of instances used for each role and the flavors, operating systems, and fragments used for define each instance.
3. The adaptation engine signals the Concord component to begin deployment.
4. The Concord component interacts with the OpenStack API to provision instances and create the necessary security rules (configure the cloud's internal firewall). The provisioning module uses a package such as *cloud-init* to initialize each cloud instance, including installing the CMT and orchestration tool agents (*e.g.*, the Puppet agent and MCollective agent).
5. Once an instance is live, the message orchestrator (*e.g.*, MCollective) prepares the instance for configuration by distributing its authentication key to the configuration manager master (*e.g.*, Puppet master).
6. The configuration is pushed to the authenticated instances using the configuration manager agent and, if needed, the orchestrator (*e.g.*, Puppet agent and MCollective).
7. System engineers may check deployed services using system monitoring applications such as Sensu [47] or Opsview [43]), or by directly accessing the instances.

In step 6, configuration is carried out via Puppet's ENC (External Node Classifier) while configuration directives (node manifests) are computed on the fly using ANCOR's operations model. This ensures that the parameters used to instantiate the Puppet classes always reflect the up-to-date system dependency information.

3.2.4 Maintenance

Once the system is deployed in the cloud, system engineers can modify the system. If the change does not affect the high-level requirement model, the maintenance is straightforward. The adaptation engine will track the impacted instances using the operations model and reconfigure them using the up-to-date system information. A good example for this type of change is cluster expansion/contraction.

Cluster expansion is used to increase the number of instances in a cluster (*e.g.*, to serve more requests or for backup purposes).

1. System engineers instruct the adaptation engine to add instances to play the role.
2. The adaptation engine triggers the Concord component to create new instances, which automatically update the ANCOR system model.
3. The adaptation engine calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.

Cluster contraction is the opposite of cluster expansion. The main goal of cluster contraction is to reduce the number of instances in a cluster (*e.g.* lowering coset and increasing efficiency).

1. System engineers instruct the adaptation engine to mark a portion of a role's instances for removal.
2. The adaptation engine calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.
3. The adaptation engine triggers the Concord component to remove the marked instances.

If the change involves major modifications in the requirement model(*e.g.,* adding/removing a role), ANCOR will need to re-compile the requirement model. We will leave this as future research.

4 Prototype Implementation

We have built a prototype (Figure 5) in Ruby [45] to implement the ANCOR framework using OpenStack as the target cloud platform. The operations model is stored in MongoDB [2] collections. ANCOR employs simple straight-forward type-checking to ensure that the requirement model is well-formed (*e.g.*, allowing a role to import a resource from another role only if the resource is exported by that role). The adaptation engine references the MongoDB document collections that store the operations model and interacts with the Concord component

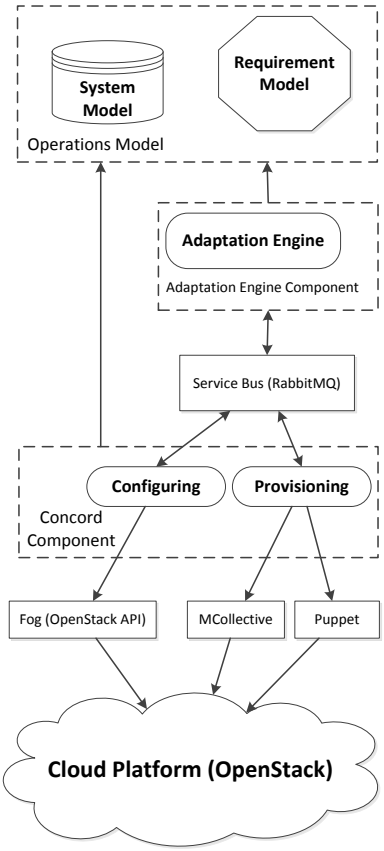

Figure 5: ANCOR Prototype Architecture

using a messaging service bus (mainly RabbitMQ [1, 5]). The Concord component interacts with the OpenStack API through Fog [41] (a cloud services library for Ruby) to provision the instances indicated by the adaptation engine. Once an instance is live, the configuration module uses Puppet and MCollective to configure it using the manifest computed on the fly based on the operations model. The Concord component also interacts with the system model and updates the provided system model database every time it performs a task (provisioning or configuration task). Therefore, the system model stored in the MongoDB datastore will always have an updated picture of the system. In the current implementation we are using a routing slip [21]-similar pattern in the communication between the Concord component and the adaptation engine. Routing slips are constructed by the adaptation engine, which oversees the whole provisioning and configuration process. We extended the service bus (RabbitMQ) using a customized plugin that acts as a router at both ends. Once a slip arrives in the service bus, the router processes the first task on the slip by extracting the handler designated to perform the task and then forwards the task to that particular handler (*i.e.,* places the task in the queue consumed by the handler). A task can be related to provisioning (*e.g.,* provision the instance using fog) or to configuring an instance (*e.g.,* push configuration from Puppet master to Puppet agent). Once the task is complete the handler reports back to the service bus and the router processes the next task on the list. We are using a thread pool in our current implementation and therefore multiple routing slips are processed in the same time (employing locks on certain shared resources).

4.1 Example Scenario

4.1.1 Initial Deployment

Assume we need to deploy and manage a scalable and highly available eCommerce website on a cloud infrastructure. We used our implemented ANCOR framework prototype to achieve this goal on an OpenStack cloud infrastructure that we deployed in our lab. Figure 6 pictures the specified requirements captured in our domain-specific language ARML.

The goal of our IT system is an eCommerce website with scalablity and high-availability. Scalability and high-availability can be achieved using a multiple-layer architecture containing various clusters of services. In deciding on the high-level structure of the overall system, we were motivated by our knowledge on what existing technologies are available in achieving the various roles. This is very realistic in a production environment, where the system engineers (with tremendous knowledge on concrete applications) define the structure of an enterprise system based on their knowledge and experience. What ANCOR would allow them to do is to be able to communicate this domain knowledge to their customers at an abstract level.

We considered the following roles to achieve the goal of the eCommerce website: web load balancer, web application, database, worker application, and messaging queue. The clustering strategy that will be employed by the applications implementing these roles may require specific role-structure to reflect the clustering strategy. There are two main clustering strategies: homogeneous and master-slave. In a homogeneous cluster all cluster members have the same configuration. In case one of the instances stops working another instance from the cluster will take over. In the master-slave strategy, different configurations are defined for the master and for the slaves. If the master fails, a slave will be promoted to the master level. In our example system, the web-balancer, web application, and the worker application employ the homogeneous strategy. Since all instances in a homogeneous cluster are configured similarly, we only need to have one role for each application. The database employs the master-slave strategy; since the two are configured differently, we need two roles for the database application: `database_master` and `database_slave`. We chose Redis to implement the messaging queue, which

```
goals:
  ecommerce:
    name: eCommerce frontend
    roles:
      - weblb
      - webapp
      - database_master
      - database_slave
      - workerapp
      - kvstore

roles:
  weblb:
    name: Web application load balancer
    min: 2
    exports:
      http:  {type:single_port, protocol:tcp}
    imports:
      webapp: http

  webapp:
    name: Web application
    min: 3
    exports:
      http:  {type:single_port, protocol:tcp}
    imports:
      database_master: querying
      database_slave: querying
      kvstore: redis

  database_master:
    name: MySQL master database
    count: 1
    exports:
      querying: {type:single_port, protocol:tcp}

  database_slave:
    name: MySQL slave database
    min: 2
    imports:
      database_master: querying
    exports:
      querying: {type:single_port, protocol:tcp}

  workerapp:
    name: Sidekiq worker application
    min: 2
    imports:
      database_master: querying
      database_slave: querying
      kvstore: redis

  kvstore:
    name: Redis server
    max: 1
    exports:
      redis: {type: single_port, protocol: tcp}
```

Figure 6: eCommerce Website Requirements Specification

does not currently support explicit clustering (web application will maintain a local buffer if the Redis will be unavailable, performance might be affected in case of very large systems but not availability). Therefore, these are the roles that we had to define in the requirement model: web load balancer (`weblb`), web application (`webapp`), `database_master`, `database_slave`, worker application (`workerapp`), and key-value store/messaging queue (`kvstore`). At this point we also specified for each role the minimum number of instances that we would like to use in the cluster.

Next we determined the dependencies (imports and exports) between the various roles. For instance, the web load balancer (`weblb`) needs to be able to load balance traffic and relay it to the web application instances. Therefore it should expose a single TCP port (http) to the web clients and import the port(s) exposed by the web application role (`webapp`); Figure 6 contains the complete role specification. Figure 7 illustrates the dependency relationship among the roles based on the specification.

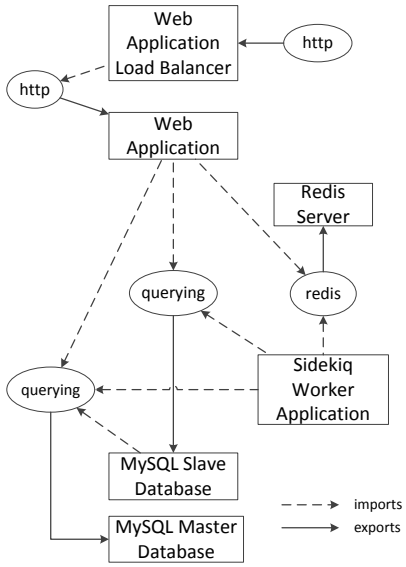

Figure 7: eCommerce Website Role Dependency (Resource Import-Export) Diagram

At this point, the system engineers likely have already had in mind what specific applications will be used to implement each role in the model. In our case, we decided that the web application role will run Ruby on Rails [6] (front-end application) with Unicorn [48] and Nginx [4] (web server). The load-balancer will run Varnish [49]. The database role will run MySQL [3] and employ a 1-to-n master-slave replication. The worker application processes background tasks (*e.g.*, order processing and emailing) that are queued by the web application and will run Sidekiq [7]. Furthermore, the messaging queue, the communication queue between the worker and web applications will be running Redis [44], which is a key-value store that can act as a queue.

It is also possible, though, that for some roles the system engineers have more than one choice to implement, perhaps with different applications with pro's and con's for each. And it may be desirable that such choices be

maintained in the requirement model to allow the ANCOR compiler to decide later based on different priorties when generating/maintaining the system. Currently we have not implemented this multi-choice mechanism in ANCOR and we intend to further investigate this in our future research. Thus for every role we wrote a single fragment as the role's implementation specification (see Section 3.1.2). The role-implementation specification binds the abstract roles to concrete building blocks provided by the configuration management system. Due to space limitation we do not include the full role-implementation specification in the paper; two example fragments were given in Figure 1 and 4. In our current prototype implementation each of those fragments is maintained in a separate file and we use a naming convention to allow ANCOR's adaptation engine to locate the implementation specification for each role by the role's name.

Once the requirements are captured and the fragments for all nodes specified, ANCOR is ready to begin the deployment process. The adaptation engine pulls the requirements and populate the operations model stored in the MongoDB. It then starts creating routing slips for every single instance. If not specified, the adaptation engine will choose random valid ports and private ip addresses. A routing slip containing the tasks for deploying an instance can be summarized in the following way:

1. Ensure network exists, if needed create it (handler can be locked if necessary)
2. Provision the instance using fog (cloud service library)
3. Update the security rules for the instances based on the system dependency (*e.g.*, open TCP port 80 for IP addresses of dependent instances)
4. Initialize the instances (key-distribution using MCollective and Puppet)
5. Initiate configuration update through Puppet

The fragments code ensure that *every time* a configuration update happens on an instance, the configuration state reflects the up-to-date system dependency information. The values of the randomly generated or chosen parameters are always stored in the system model (part of the operations model). And those values are used in the ERB code in the fragments to generate the configuration state to be sent to the instances.

Once the initial deployment process was completed, we were to check the functionality of the deployed eCommerce website.

4.1.2 Management and Evaluation

The testbed we used was built in the Argus cybersecurity lab at Kansas State University and consists of three hosts with i3-2100T CPU@2.5GHz with 4GB of RAM, and a Dell PowerEdge T620 tower server (2xIntel Xeon E5-2660 2.20GHz, 200GB Solid State Drive and 64 GB RAM). We deployed a four-node OpenStack Grizzly infrastructure on these machines: controller node, network node, compute node A - i3-2100T machines and compute node B - Dell T620. We used Apache JMeter (benchmarking tool for web applications, servers *etc.* [39]) to test the system's availability and performance while managing various clusters (*i.e.,* adding/removing instances to a cluster). We ran the benchmarking tool on the initially deployed eCommerce website and established a baseline for every component in our measurements. JMeter ran from a client machine that connects to the eCommerce website (*i.e.,* connects to the load balancer) and for each test we sent a total of 100,000 requests using 10 threads. Furthermore, we targeted various links that ran operations we implemented to specifically test the various system components (*e.g.*, read from the database slaves).

	Failure Rate (%)	Average Response Time (ms)
Baseline	0.00%	25ms
Adding	0.02%	33ms
Removing	0.02%	33ms

Table 1: Benchmarking - `database_slave` cluster

Table 1 shows the benchmarking result for reading data through the full stack of the applications (from the load balancer to the database slaves). After establishing the baselines, we started adding and removing instances to and from different clusters while targeting operations that involve the deepest cluster in the stack (database slave). When adding and removing instances to and from the MySQL slaves cluster, performance and availability might be slightly affected. On average we experienced failure rate of 0.02%. In other words, out of 100,000 sent requests, 2000 timed out or an HTTP error was recorded. Furthermore, the average response time slightly increased, from 25 ms to 33 ms. All benchmarking results are influenced by the way applications were configured (*e.g.*, "hot-swap" feature). The "hot-swap" feature loads an updated configuration without restarting a service (*e.g.*, Unicorn). This is not caused by ANCOR and could be reduced by tuning the applications. The rest of the components exhibited the same performance and high-availability trends like the database slaves.

5 Discussion and Future Work

Our ANCOR prototype shows that capturing user requirements in an abstract specification and compiling it into an operational cloud-based system is not only possible, but has the potential to fundamentally change the

way cloud-based systems are created and managed. We chose the term *compile* to describe our translation of the abstract specification into a real system as we believe it is similar in fashion and function to the modern-day code compilers common to every programmer. In the early days of computing, machine code was placed into computer memories much the way configurations are manually created and placed on hosts. The assembler, much like current cloud management tools, made doing this simpler and less error-prone. However, it was the advent of the compiler that allowed programmers to move to the higher level of abstraction found in programming languages such as Cobol or FORTRAN that really accelerated the growth of the computer industry. There can be no doubt that the Internet would not exist in its current form if we were still programming in assembly code. In the same way as code compilers revolutionized software development, we believe that *system compilers* can revolutionize cloud-based systems development and management.

The high-level requirement model we developed could also facilitate tasks like failure diagnosis and system analysis to identify design weaknesses such as single point of failures or performance bottlenecks. This is not unlike code debugging and static analysis of programs where a high-level language (source code) could significantly ease the tasks. The system dependency information specified in the requirement model and maintained in the operations model allows for more reliable and streamlined system update such as service patching. It also allows for more fine-grained firewall setup (*i.e.*, only allows network access that is consistent with the system dependency), and enables porting systems to a different cloud providers in a more organized manner (*e.g.*, one can take the up-to-date requirement model and compile it to a different cloud provider's infrastructure, and then migrate data from the old one to the new one).

We are continuing to develop the ANCOR framework by adding more features to the implementation. One of our priorities is to construct a pro-active change mechanism as part of the adaptation engine. This mechanism would randomly select specific aspects of the configuration to change (*e.g.*, virtual machine addresses, application ports, software versions, *etc.*); the changes would be automatically carried out by the Concord component. Our goal in this is to analyze the possible security benefits as well as to measure and analyze the performance, robustness, and resilience of such a mechanism. We are also planning on automating the use of the requirement model for automated root cause analysis and diagnosis of faults. Furthermore, we are looking into replacing the current routing slip workflow model with a parallel workflow model (similar to the lifecycle events in AWS OpsWorks) to support the use of multiple configuration management tools such as Chef. In addition, we are planning on switching to the OpenStack AWS compatible API that will enable us to use the same implementation on different cloud platforms.

We intend to release the first version of the ANCOR framework in the near future (in a few months) under an open-source license.

6 Related Work

Cloud computing has the potential to be one of the revolutionary technologies in IT history [9]. However, since it is still relatively new, simply defining the terminology [9, 29] and understanding the current state of technology [33] related to cloud computing has been a major topic of interest. In addition, there has been significant discussion of the the technical, economic, and legal implications of using cloud services for key infrastructures [9, 14, 15, 20, 23, 35]. While many benefits appear to exist, the discovery of a variety of potential vulnerabilities has raised significant concerns (*e.g.,* [11] and [13]). This has lead to research on solutions that address issues in security and efficiency (*e.g.,* [12, 36, 50]). Here, past work on high-availability and security in virtual environments [22, 24] can inform cloud research as well. Another area of concern that has garnered significant research is preserving the integrity and privacy of user data stored in the cloud [15, 18, 19, 25, 28, 31, 37, 38]. We believe our work in ANCOR will help address some of the manageability and security issues raised by prior research, by making cloud-based IT systems easy to create and manage and leaving less room for human errors.

Currently network configuration tends to be static and routine assumptions are made about the whereabouts of services in terms of either fixed URL's or IP addresses. Past work has examined service dependencies in large enterprise environments and found that the dependencies among services are often surprising [10, 16]. This is due to lack of documentation of an IT system's structure and overly relying upon assumptions made when developers/system administrators hard-coded dependencies into programs or configuration files. The lack of a way to maintain the dependency information makes managing services more difficult since it is unclear whether a change to one service will disrupt other services due to unexpected dependencies. Our work in ANCOR will solve this problem by making the service dependencies explicit in the requirement model and maintained accurately all the time.

Albrecht et al. [8] introduced Plush, a generic application management infrastructure that provides a unified set of abstractions for specifying, deploying, and monitoring distributed applications (*e.g.,* peer-to-peer services, web search engines, social sites, *etc.*). While Plush's architecture is flexible, it is not targeted at de-

ploying enterprise systems on cloud infrastructures, and it is not clear whether system dependencies can be specified in a straightforward way and maintained throughout the lifecycle of the deployed system.

The idea of designing a higher-level abstraction to better manage a system has also been investigated in the context of Software-Defined Networking (SDN). Monsanto *et al.* [27] introduced abstractions for building applications out of multiple, independent modules that jointly manage network traffic. The Pyretic language and system enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract network topologies. Our ANCOR langauge and system adopts the same philosophy and we focus on abstraction for cloud system deployment and management.

7 Conclusion

Separating user requirements from the implementation details has the potential of changing the way IT systems are deployed and managed in the cloud. To capture user requirements, we developed a high-level abstraction called the Requirement Model for defining cloud-based IT systems. Once users define their desired system in the specification, it is automatically compiled into a concrete cloud-based system that meets the specified user requirements. We demonstrate the practicality of this approach in the ANCOR framework. Preliminary benchmarking results show that ANCOR can improve manageability and maintainablity of cloud-based system and enable dynamic configuration changes of a deployed system with negligible performance overhead. While the current implementation of ANCOR uses Puppet to configure OpenStack cloud instances, we are confident that ANCOR could work with other cloud platforms and configuration management solutions as well. We are planning to release the first version of the ANCOR framework by February 2014 under an open-source license.

A Appendix

Figure 8 shows the complete entity-relation diagram for the operations model. The arrows indicate the direction of references in the implementation (one-way or two-way references) and the associated multiplicity (1-to-1, 1 to n, or n-to-n). For instance, one role may support more multiple goals, and multiple roles could support one goal. Thus the multiplicity between goal and role is n-to-n.

The system model is a local reflection of the cloud and, as previously mentioned, it is used for bookkeeping. This enables the user to track instances in terms of roles. Furthermore, the system model bridges the gap between the more abstract requirement model and the many different

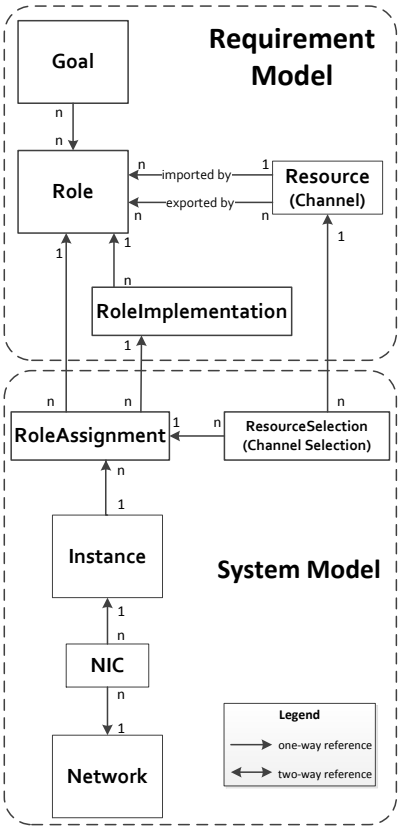

Figure 8: Operations Model

concrete systems that can implement it (see Figure 2). The requirement model is considered read-only by the rest of the framework. On the other hand, the system model can be updated by every component in the framework.

An *instance* is a virtual machine that a role is assigned to. A role can be assigned to more than one instance and an instance can fulfill multiple roles. *RoleAssignment* captures the relationship between a role and an instance. Furthermore, a role can have one or more ways of being implemented. This aspect is captured in *RoleImplementation*. In other words, this is a way to inform the configuration manager tool on the implementation side what transformations to apply to an instance (*i.e.,* a way to tie configuration manager tool setup modules to an instance). A *NIC* keeps the MAC address(es) that belong to an instance and a *Network* stores the network(s) that an instance is connected to.

References

[1] AMQP website - accessed 9/2013. http://www.amqp.org/.
[2] MongoDB website - accessed 9/2013. http://www.mongodb.org/about/introduction/.
[3] MySQL website - accessed 9/2013. http://www.mysql.com/.
[4] Nginx website - accessed 9/2013. http://nginx.org/.

[5] RabbitMQ website - accessed 9/2013. http://www.rabbitmq.com/.

[6] Ruby on Rails website - accessed 9/2013. http://rubyonrails.org/.

[7] Sidekiq Repository on GitHub - accessed 9/2013. https://github.com/mperham/sidekiq.

[8] J. Albrecht, C. Tuttle, R. Braud, D. Dao, N. Topilski, A.C. Snoeren, and A. Vahdat. Distributed Application Configuration, Management, and Visualization with Plush. *In ACM Transactions on Internet Technology (Volume: 11, Issue: 2, Article No. 6)*, 2011.

[9] M. Armbust, A. Fox, R Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, pages 50–58, 2010.

[10] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM 2007*, 2007.

[11] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A Security Analysis of Amazon's Elastic Compute Cloud Service. *In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1427–1434, 2012.

[12] A. Benameur, N.S. Evans, and M.C. Elder. Cloud Resilience and Security via Diversified Replica Execution. *In Proceedings of 1st international Symposium on Resilient Cyber Systems*, 2013.

[13] S. Bugiel, S. Nürnberger, T. Pöppelman, A. Sadeghi, and T. Schneider. AmazonIA: When Elasticity Snaps Back. *In Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, pages 389–400, 2011.

[14] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyy. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *In Proceedings of Software: Practice and Experience (Volume 41, Issue 1)*, pages 23–50, 2011.

[15] D. Chen and H. Zhao. Data Security and Privacy Protection Issues in Cloud Computing. *In Proceedings of the International Conference on Science and Electronics Engineering (ICCSEE)*, pages 647–651, 2012.

[16] Xu Chen, Ming Zhang, Z. Morley Mao, and Victor Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, 2008.

[17] C. C. Evans. YAML - accessed 8/2013. http://yaml.org/.

[18] A.J. Feldman, A. Blankstein, M.J. Freedman, and E.W. Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. *In Proceedings of the 21st USENIX Security Symposium*, 2011.

[19] A.J. Feldman, W.P. Zeller, M.J. Freedman, and E.W. Felten. Sporc: Group Collaboration using Untrusted Cloud Resources. *In Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[20] M. Hajjat, X. Sun, Y.E Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. *In Proceedings of the ACM SIGCOMM Conference*, pages 243–254, 2010.

[21] G. Hohpe and B. Woolf. Chapter 7 - Message Routing. In *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., 2003.

[22] Y. Huang, D. Arsenault, and A. Sood. Incorruptible System Self-Cleansing for Intrusion Tolerance. *In Proceedings of 25th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 486–496, 2006.

[23] Y. Huang, D. Arsenault, and A. Sood. Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control. *In Proceedings of the ACM workshop on Cloud Computing Security*, pages 85–90, 2009.

[24] Yih Huang, Arun Sood, and Ravi K. Bhaskar. Countering Web Defacing Attacks with System Self-Cleansing. *Proceedings of 7th Word Multiconference on Systemics, Cybernetics and Informatics*, pages 12–16, 2003.

[25] E. Keller, D. Drutskoy, J. Szefer, and J. Rexford. Cloud Resident Data Center. Technical report, Princeton University, 2011.

[26] Puppet Labs. What is Puppet? - accessed 9/2013. http://puppetlabs.com/puppet/what-is-puppet.

[27] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. *In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2013.

[28] M. Mowbray, S. Pearson, and Y. Shen. Enhancing Privacy in Cloud Computing via Policy-based Obfuscation. *In The Journal of Supercomputing (Volume 61, Issue 2)*, pages 267–291, 2012.

[29] NIST. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, 2011.

[30] Opscode. Chef - accessed 9/2013. http://www.opscode.com/chef/.

[31] S. Pearson and A. Benameur. Privacy, Security and Trust Issues Arising from Cloud Computing. *In Proceedings of IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 693–702, 2010.

[32] RightScale. White Paper - Quantifying the Benefits of the RightScale Cloud Management Platform - accessed 8/2013. http://www.rightscale.com/info_center/white-papers/RightScale-Quantifying-The-Benefits.pdf.

[33] Service-now.com. White Paper - Managing the Cloud as an Incremental Step Forward - accessed 8/2013. http://www.techrepublic.com/resource-library/whitepapers/managing-the-cloud-as-an-incremental-step-forward/.

[34] Amazon Web Services. AWS OpsWorks - accessed 9/2013. http://aws.amazon.com/opsworks/.

[35] A. Sheth and A. Ranabahu. Semantic Modeling for Cloud Computing. *In Proceedings of Internet Computing, IEEE (Volume:14 , Issue: 4)*, 2010.

[36] J. Szefer, E. Keller, R.B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface For a More Secure Cloud. *In Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, pages 401–412, 2011.

[37] C. Wang, Q. Wang, K. Ren, and W Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. *In Proceedings of INFOCOM*, pages 1–9, 2010.

[38] C. Wang, Q. Wang, K. Ren, and W Lou. Privacy-Preserving Public Auditing for Secure Cloud Storage. *In Proceedings of IEEE Transactions on Computers (Volume:62 , Issue: 2)*, pages 362–375, 2013.

[39] Apache JMeter Website. JMeter - accessed 9/2013. http://httpd.apache.org/docs/2.2/programs/ab.html.

[40] CFEngine Website. What is CFEngine? - accessed 9/2013. http://cfengine.com/what-is-cfengine.

[41] Fog Website. The Ruby Cloud Services Library - accessed 9/2013. http://fog.io/.

[42] OpenStack Website. Open Source Software for Building Private and Public Clouds - accessed 9/2013. http://www.openstack.org/.

[43] OpsView Website. OpsView - accessed 9/2013. http://www.opsview.com/.

[44] Redis Website. Redis - accessed 9/2013. http://redis.io/.

[45] Ruby Website. Ruby is... - accessed 9/2013. https://www.ruby-lang.org/en/.

[46] SaltStack Website. What is SaltStack? - accessed 9/2013. http://saltstack.com/community.html.

[47] Sensu Website. Sensu - accessed 9/2013. http://sensuapp.org/.

[48] Unicorn! Website. What it is? - accessed 9/2013. https://github.com/blog/517-unicorn.

[49] Varnish Cache Website. About - accessed 9/2013. https://www.varnish-cache.org/about.

[50] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing Security of Virtual Machine Images in a Cloud Environment. *In Proceedings of the 2009 ACM workshop on Cloud Computing Security*, pages 91–96, 2009.